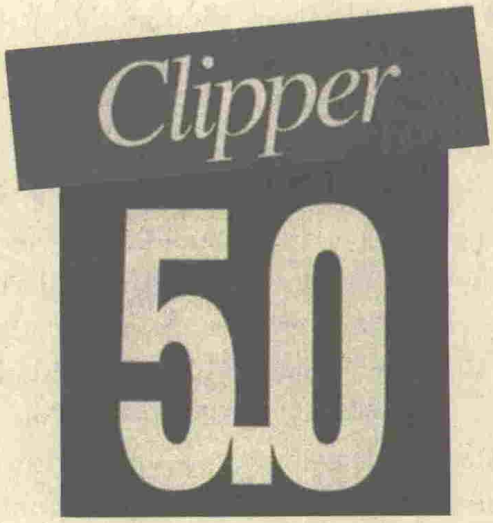
จเร เลิศสุดวิชัย

การใช้งานตัวแปรในคลิปเปอร์ 5.0

ตอนที่ 2



ศึกษาการใช้ตัวแปรระบบ ที่เป็นสวิตช์ และเครื่องหมายการกระทำกับนิพจน์ รวมถึงนิพจน์เงื่อนไข และรูปแบบคำสั่งต่าง ๆ

ก่อนจะเข้าถึงเรื่องของตัวแปรคลิปเปอร์ในเรื่องขอบเขต (Scope) และช่วงเวลาชีวิต (Lifetime) ก็จะกล่าวถึงเรื่องที่เกี่ยวข้องและมีความสำคัญเป็นพื้นฐานก่อนดังนี้

คำสั่ง SET

ในคลิปเปอร์มีตัวแปรระบบอยู่หลายตัวที่เป็นสวิตช์และมีผลต่อการทำงานของโปรแกรมเหมือนดีเบส ซึ่งสวิตช์เหล่านั้นสามารถสั่งให้เป็น on หรือ off ได้ แต่ในคลิปเปอร์นั้นมีฟังก์ชันที่ใช้สำหรับจัดการกับตัวแปรสวิตช์เหล่านี้ได้โดยตรง (นอกจากการสั่งด้วยคำสั่ง SET) โดยมีวิธีการใช้งานดังนี้

```
SET (<หมายเลขสวิตช์>, [ค่าใหม่ที่ต้องการเปลี่ยน])
```

และฟังก์ชัน SET จะรีเทอร์นค่าเดิมของสวิตช์นั้นกลับคืนมา ในคู่มือแนะนำว่า หมายเลขสวิตช์นั้นจะไม่ตรงกันในแต่ละเวอร์ชันของคลิปเปอร์ ดังนั้นให้ใช้ชื่อที่กำหนดเป็นค่าคงที่ในไฟล์ของ SET.CH แทน เช่น

สวิตช์ของ EXACT จะถูกกำหนดชื่อให้เป็น 1 ดังนี้

```
#define __SET__EXACT 1
```

เช่น ถ้าต้องการอ่านค่าของสวิตช์ EXACT มาเก็บไว้ก็สามารถทำได้

```
cur__exact = set (__SET__EXACT)
```

ในขณะที่ทำการคอมพายล์โปรแกรมนั้น คลิปเปอร์จะอ่านเอาไฟล์ SET.CH เข้ามาร่วมในการคอมพายล์ให้โดยอัตโนมัติอยู่แล้ว จึงไม่ต้องสั่ง

```
#include "Set.ch"
```

อีกก็ได้

ในรูปที่ 1 เป็นตารางแสดงสวิตช์ต่าง ๆ ตามค่าปกติที่กำหนดไว้ (default) และชื่อของสวิตช์ที่สามารถนำมาใช้แทนตัวเลขได้

จากวิธีการใช้งานฟังก์ชัน SET เราสามารถจะเปลี่ยนค่าของสวิตช์ใหม่ได้ โดยส่งเป็นพารามิเตอร์ตัวที่สองต่อจากชื่อของสวิตช์ลงไปในการเรียกใช้งานฟังก์ชัน SET ซึ่งฟังก์ชัน SET ก็จะนำเอาค่าใหม่ดังกล่าวไปใส่ไว้ในตัวแปรสวิตช์ของระบบและรีเทอร์นค่าเดิมออกมา

สวิตช์	ค่าที่กำหนดให้ตามปกติ	ชื่อสวิตช์	คำสั่งหรือฟังก์ชันที่เกี่ยวข้อง
ALTERNATE	OFF	__SET__ALTERNATE	SET ALTERNATE
ALTERNATE TO	CLOSED	__SET__ALTFILE	SET ALTERNATE TO
BELL	OFF	__SET__BELL	SET BELL
COLOR	W/N, N/W, N/N N/N, N/W	__SET__COLOR	SETCOLOR ( )
CONFIRM	OFF	__SET__CONFIRM	SET CONFIRM
CONSOLE	ON	__SET__CONSOLE	SET CONSOLE
CURSOR	ON	__SET__CURSOR	SETCURSOR ( )
DATE	AMERICAN	__SET__DATEFORMAT	SET DATE
DECIMALS	2	__SET__DECIMALS	SET DECIMALS
DEFAULT	(NONE)	__SET__DEFAULT	SET DEFAULT
DELETED	OFF	__SET__DELETED	SET DELETED
DELIMITERS	OFF	__SET__DELIMITERS	SET DELIMITERS
DELIMITERS TO	LEFT: RIGHT:	__SET__DELIMCHARS	SET DELIMITERS TO
DEVICE	SCREEN	__SET__DEVICE	SET DEVICE
EPOCH	1900	__SET__EPOCH	SET EPOCH
ESCAPE	ON	__SET__ESCAPE	SET ESCAPE
EXACT	OFF	__SET__EXACT	SET EXACT
EXCLUSIVE	ON	__SET__EXCLUSIVE	SET EXCLUSIVE
FIXED	OFF	__SET__FIXED	SET FIXED
INTENSITY	ON	__SET__INTENSITY	SET INTENSITY
MARGIN	0	__SET__MARGIN	SET MARGIN
MESSAGE	0	__SET__MESSAGE	SET MESSAGE
MESSAGE TO...CENTER	.F.	__SET__MCENTER	SET MESSAGE
PATH	ON	__SET__PATH	SET PATH
PRINT	OFF	__SET__PRINTTER	SET PRINTTER
PRINT TO		__SET__PRINTFILE	SET PRINT TO
SCOREBOARD	ON	__SET__SCOREBOARD	SET SCOREBOARD
SOFTSEEK	OFF	__SET__SOFTSEEK	SET SOFTSEEK
TYPEAHEAD	15	__SET__TYPEAHEAD	SET TYPEAHEAD TO
UNIQUE	OFF	__SET__UNIQUE	SET UNIQUE
WRAP	OFF	__SET__WRAP	SET WRAP

รูปที่ 1 ตารางแสดงสวิตช์ต่าง ๆ ของตัวแปรระบบ

```
PRIVATE save__sets [__SET__COUNT]
  FOR i = 1 TO __SET__COUNT
    save__sets [i] = set (i)
NEXT
```

รูปที่ 2

```
? "Clipper 5.0" = "Clipper" จะให้ค่าเป็นจริง (.T)
? "Clipper" = "Clipper 5.0" จะให้ค่าเป็นเท็จ (.F.)
```

รูปที่ 3

ดังนั้นเราสามารถจะทำการเก็บเอาค่าเดิมของตัวแปรสวิตช์ทั้งหมดเอาไว้ใช้งานในโอกาสต่อไปได้ดังรูปที่ 2

โดยในไฟล์ SET.CH จะมีค่า __SET__COUNT ซึ่งแสดงถึงจำนวนของสวิตช์ที่มีทั้งหมดในคลิปเปอร์

ถึงตรงนี้หลายคนอาจจะสงสัยบ้างว่าทำไมถึงใช้ตัวแปรอะเรย์เก็บค่าของตัวแปรสวิตช์ต่าง ๆ ได้ ทั้งที่ค่าของตัวแปรสวิตช์แต่ละตัวไม่เหมือนกัน ซึ่งบางตัวนั้นเป็นค่าลอจิก (Logic) บางตัวก็เป็นอักขระ (Character) อันนี้เป็นเรื่องพิเศษของตัวแปรอะเรย์ที่เพิ่มเติมขึ้นมาในคลิปเปอร์ 5.0 ซึ่งรายละเอียดจะกล่าวถึงอีกครั้งในเรื่องอะเรย์

SET EXACT

สวิตช์ตัวนี้มักจะสร้างความสับสนในการใช้งานอยู่เสมอ ดังนั้นก่อนจะข้ามไปเรื่องอื่นก็จะมาพิจารณาเรื่องนี้ดูอย่างละเอียดก่อน

สวิตช์ EXACT จะมีผลต่อการเปรียบเทียบข้อความด้วยเครื่องหมายเท่ากับ (=) ตามปกติสวิตช์ EXACT จะมีค่าเป็น OFF ดังนั้น ผลการเปรียบเทียบจะให้เป็นจริงก็ต่อเมื่อ นำข้อความที่อยู่ทางด้านขวาของเครื่องหมายเท่ากับ ตั้งแต่ตัวแรกจนถึงตัวสุดท้าย ไปเปรียบเทียบกับข้อความที่อยู่ทางด้านซ้ายของเครื่องหมายเท่ากับตั้งแต่ตัวแรกจนได้ความยาวเท่ากับข้อความที่อยู่ด้านขวา แล้วปรากฏว่าข้อความเหมือนกันทุกตัวก็จะให้ค่าเป็นจริง นั้นหมายถึงหากว่าข้อความทางซ้ายมือของเครื่องหมายเท่ากับยังมีต่ออีก ก็จะไม่มีผลต่อการเปรียบเทียบ ตัวอย่างเช่นรูปที่ 3

ในขณะที่สวิตช์ EXACT ยังมีสถานะเป็น OFF อยู่ มักจะสร้างปัญหาให้กับผู้เริ่มเขียนโปรแกรมด้วยดีเบสหรือคลิปเปอร์เสมอ เมื่อมีการใช้งานกับฟิลด์ของแฟ้มข้อมูลดังรูปที่ 4

การเปรียบเทียบในตัวอย่างในรูปที่ 4 จะให้ผลลัพธ์เป็นเท็จ ถึงแม้ว่าข้อความทั้งด้านซ้ายและด้านขวาของเครื่องหมายเท่ากับใน 7 ตัวแรกนั้นจะ

ตรงกัน แต่ข้อความทางขวามือนั้นยังมีช่องว่าง (space) ต่ออีก 3 ช่อง (ตามความกว้างของฟิลด์ที่ใช้เก็บข้อมูล) ซึ่งถือว่าข้อความทั้งสองข้างนั้นไม่เหมือนกันจึงให้ค่าเป็นเท็จ

อีกตัวอย่างหนึ่ง ซึ่งมักจะเกิดขึ้นโดยที่ผู้เขียนโปรแกรมไม่ค่อยได้สังเกตนักคือ การตรวจสอบตัวแปรข้อความที่ว่าง (NULL) ดังนี้

```
#define NULLC ""
.
.
.
IF name = NULLC
```

ในขณะที่สวิตช์ EXACT มีสถานะเป็น OFF อยู่นี้ ถ้าหากว่า ตัวแปร name มีข้อความบรรจุอยู่ (เช่น อาจจะเก็บข้อความว่า "Clipper") ผลของการเปรียบเทียบก็จะได้ค่าจริงเสมอ ซึ่งเป็นสิ่งที่ไม่ถูกต้อง ดังนั้นถ้าต้องการให้การทำงานถูกต้องก็ให้สลับตำแหน่งของตัวแปรในการเปรียบเทียบเสียดังนี้

```
IF NULLC = name ซึ่งจะได้ค่าเท็จ
```

ซึ่งเป็นผลลัพธ์ที่ถูกต้องตรงตามความต้องการ

ข้อควรระวังอีกเรื่องหนึ่งในกรณีที่มีการใช้งานกับแฟ้มข้อมูล โดยต้องการจะลบข้อมูลในเรคอร์ดที่มีค่าตรงกับที่ต้องการ โดยใช้คำสั่งดังนี้

```
DELETE FOR upper(name) = "CLIPPER"
```

คำสั่งนี้จะทำการลบข้อมูลในทุกเรคอร์ดที่มีฟิลด์ name บรรจุข้อความที่ขึ้นต้นด้วย "CLIPPER" เช่น เรคอร์ดที่บรรจุข้อความว่า

```
"CLIPPER SUMMER '87"
```

```
name1 = names_dbf→name   && เป็นฟิลด์ชนิดตัวอักษรกว้าง 10 ช่องและบรรจุข้อความว่า "Clipper  "
name2 = "Clipper"
? name2 = name1
```

รูปที่ 4

```
SET EXACT ON
name1 = names_dbf→name   && เป็นฟิลด์ของข้อความขนาด 10 ช่อง เก็บข้อความว่า "Clipper  "
name2 = "Clipper"
? name2 = name1
```

รูปที่ 5

หรือ

```
"CLIPPER 5.0"
```

หรือ

```
"CLIPPER 5.01"
```

และในกรณีที่สวิตช์ EXACT มีสถานะเป็น ON การเปรียบเทียบข้อความที่อยู่ทางด้านซ้ายและด้านขวาของเครื่องหมายเท่ากับจะให้ค่าเป็นจริงก็ต่อเมื่อ ข้อความทั้งสองข้างต้องตรงกันทุกตัวอักษร แต่ไม่สนใจช่องว่างที่ตามหลังข้อความ ดังรูปที่ 5

ผลการเปรียบเทียบนี้จะได้ผลลัพธ์เป็นค่าจริง (.T.) ซึ่งในการเปรียบเทียบกรณีที่สวิตช์ EXACT เป็น ON จะไม่สนใจช่องว่าง (space) ที่ต่อท้ายข้อความแต่จะสนใจเฉพาะข้อความที่เป็นตัวอักษรเท่านั้น ถ้าเหมือนกันทุกตัวก็จะให้ค่าเป็นจริง

อย่างไรก็ตาม ในเวลาใช้งานจริง ๆ การเปรียบเทียบทั้งแบบจริงจัง (EXACT ON) และแบบไม่จริงจัง (EXACT OFF) ก็มีประโยชน์ทั้งสองอย่างขึ้นอยู่กับความต้องการหรือวัตถุประสงค์ในการทำงานนั้น ๆ เช่น บางครั้งเราก็ต้องการค้นหาข้อมูลที่ระบุค่าอย่างเจาะจง ลักษณะเช่นนี้เราต้องกำหนดให้ EXACT มีสถานะเป็น ON หรือบางครั้งเราต้องการค้นหากลุ่มของข้อมูลใด ๆ โดยระบุเจาะจงเฉพาะตัวอักษรที่ขึ้นต้นเท่านั้น ว่าต้องเป็นตัวอักษรตามที่เรากำหนด ส่วนตัวอักษรอื่น ๆ ที่ต่อท้ายจะเป็นอะไรก็ได้เราไม่สนใจ ลักษณะเช่นนี้เราจะต้องกำหนดให้ EXACT มีสถานะเป็น OFF ดังนั้นการปรับสวิตช์ EXACT จะต้องทำด้วยความระมัดระวัง เพราะอาจจะทำให้โปรแกรมที่เราเขียนขึ้นใช้งานนั้นทำงานผิดพลาด ไม่เป็นไปตามความต้องการของเราก็ได้

ในคลิปเปอร์ 5.0 ได้เพิ่มเครื่องหมายในการกระทำการเปรียบเทียบขึ้นมาอีกเครื่องหมายหนึ่ง คือ "=="

โดยเครื่องหมายนี้ถ้าใช้ในการเปรียบเทียบข้อความแล้ว ข้อความทั้งสองข้างของเครื่องหมายนี้จะต้องมีความเหมือนกันทุกประการ กล่าวคือ จะต้องเหมือนกันทั้งขนาดความยาวของข้อความ และตัวอักษรทุกตัวในข้อความทั้งสองข้างต้องตรงกันด้วย นั่นหมายถึงว่า ถ้าหากมีช่องว่าง (space) ต่อท้ายข้อความก็จะต้องมีเหมือนกันทั้งสองข้าง และจำนวนของช่องว่างต้องเท่ากันด้วย จึงจะได้ผลลัพธ์ว่าเท่ากันจริง (ซึ่งไม่เหมือนกับกรณีที่ใช้สวิตช์ EXACT เป็น ON อยู่ เพราะจะไม่สนใจช่องว่าง)

เครื่องหมาย "==" ถ้าหากใช้กับตัวแปรอะเรย์จะเป็นการตรวจสอบว่า ตัวแปรอะเรย์ทั้งสองตัวนั้นเป็นตัวแปรตัวเดียวกันหรือไม่ (คืออยู่ในหน่วยความจำตำแหน่งเดียวกันหรือเปล่า)

เครื่องหมายการกระทำกับนิพจน์

ในคลิปเปอร์จะมีเครื่องหมายการกระทำทางคณิตศาสตร์ แบบไบนารีอยู่ดังนี้

```
** ^
* / %
+ -
```

โดยจะจัดลำดับความสำคัญในการคำนวณตามกลุ่มแต่ละบรรทัดจากบรรทัดบนสุด ซึ่งจะมีลำดับในการคำนวณก่อนสุด นั่นคือ เครื่องหมาย ** และ ^ จะถูกคำนวณก่อนแล้วจึงตามด้วยเครื่องหมายในกลุ่ม *, / และ % และต่อด้วยกลุ่ม + และ - โดยที่เครื่องหมายแต่ละกลุ่มนั้น จะมีลำดับการคำนวณเท่ากัน ดังนั้นคลิปเปอร์จะทำการคำนวณในนิพจน์จากทางซ้ายไปขวา

ในคลิปเปอร์ยังมีเครื่องหมายค่าลบ "-" ซึ่งเป็นเครื่องหมายแบบ unary ซึ่งมีลำดับการคำนวณก่อนสุด โดยมีลำดับการคำนวณก่อนกลุ่มเครื่องหมายทางคณิตศาสตร์ข้างต้นด้วย เช่น

```
? -1+2
```

จะให้ค่าเป็น 1 ไม่ใช่ -3

นอกจากนั้นในคลิปเปอร์ยังมีเครื่องหมายการเพิ่มค่า และลดค่าเหมือนที่มีใช้อยู่ในภาษาซีอีกด้วย คือ เครื่องหมาย ++ และ --

โดยที่เครื่องหมาย ++ หมายถึงบวกเพิ่มด้วย 1 และเครื่อง -- หมายถึง ลบออกด้วย 1 ดังตัวอย่าง

```
counter + + จะเหมือนกับ counter = counter + 1
```

ข้อดีของเครื่องหมายทั้งสองคือสามารถใช้งานในนิพจน์ได้ ทำให้การเขียนโปรแกรมนั้นสั้นลง ดังตัวอย่าง ถ้าต้องการดึงเอาข้อมูลจากฟิลด์ของ name จากทุก ๆ เรคอร์ดในแฟ้มข้อมูลมาเก็บไว้ในตัวแปรอะเรย์ ก็สามารถทำได้โดยใช้เครื่องหมาย ++ ช่วยดังรูปที่ 6

```
PRIVATE name [reccount ( )]
counter = 1
USE sales
Do WHILE ! eof ( )
   name [counter + + ] = name
   SKIP
ENDDO
```

รูปที่ 6

ในกรณีนี้เครื่องหมาย ++ ถูกวางอยู่ท้ายตัวแปร counter ดังนั้นหลังจากที่ได้นำค่าจากฟิลด์ name ไปเก็บไว้ในตัวแปรอะเรย์ names ช่องที่ counter แล้วค่าใน counter จึงจะถูกเพิ่มค่าขึ้นอีก 1

ซึ่งจะเห็นว่าในกรณีที่ใช้เครื่องหมาย ++ หรือ -- วางไว้หลังตัวแปรในประโยคนิพจน์ หลังจากเสร็จสิ้นการทำงานในนิพจน์ดังกล่าวแล้ว (โดยใช้ค่าในตัวแปรที่มีอยู่เดิมก่อน) ค่าในตัวแปรนั้นจึงจะถูกเพิ่มค่าหรือลดค่าลงอีก 1 (การเพิ่มค่าหรือลดค่าเกิดขึ้นภายหลังการทำงานของนิพจน์นั้น)

นอกจากนี้การใช้เครื่องหมาย ++ หรือ -- ยังสามารถนำมาวางไว้ด้านหน้าของตัวแปรก็ได้ แต่การทำงานจะไม่เหมือนกับกรณีที่วางไว้หลังตัวแปร กล่าวคือ ค่าของตัวแปรจะถูกเพิ่มค่าหรือลดค่า (ตามเครื่องหมาย) ด้วย 1 ก่อน จึงจะนำไปใช้งานในนิพจน์นั้น (ค่าของตัวแปรเป็นค่าใหม่) ดังตัวอย่างข้างต้น ถ้าต้องการใช้เครื่องหมาย ++ วางไว้หน้าตัวแปร counter จะเขียนได้ใหม่ดังรูปที่ 7

นอกจากนั้นคลิปเปอร์ยังสนับสนุนเรื่องการใช้เครื่องหมายในการกำหนดค่าแบบผสม (assignment) อีกด้วย ซึ่งก็นำมาจากในภาษาซีเช่นเดียวกัน ดังนี้

```
PRIVATE name [reccount ( )]
counter = 0
USE sales
DO WHILE ! eof ( )
      name [+ + counter] = name
      SKIP
ENDDO
```

รูปที่ 7

```
+=
-=
*=
/=
%=
**= หรือ . ~=
```

ตัวอย่างเช่น

```
i += 10 ก็คือ i = i + 10
```

เครื่องหมายในการกำหนดค่าแบบผสมนั้น ช่วยทำให้การเขียนโปรแกรมสั้นลง

สำหรับเครื่องหมายในการเปรียบเทียบ ซึ่งมีดังนี้คือ

```
<   ><=   >= = != = =
```

เครื่องหมายในกลุ่มนี้จะมีลำดับการคำนวณเท่าเทียมกัน และคลิปเปอร์จะทำการคำนวณจากทางซ้ายไปทางขวาของนิพจน์ ถ้ามีเครื่องหมายเหล่านี้ประกอบอยู่

ในกลุ่มของเครื่องหมายการเปรียบเทียบจะมีลำดับการคำนวณต่ำกว่ากลุ่มของเครื่องหมายทางคณิตศาสตร์ ตัวอย่างเช่น

```
i < limit -1
```

จะมีการคำนวณตามลำดับดังนี้ i<(limit-1) ไม่ใช่ (i<limit)-1

และในกลุ่มเครื่องหมายกระทำทางลอจิก คือ

```
.AND. .OR. และ .NOT.
```

(ในกรณีของเครื่องหมาย .NOT. สามารถใช้เครื่องหมาย ! แทนได้)

เครื่องหมายในกลุ่มนี้จะมีลำดับการคำนวณต่ำกว่ากลุ่มเครื่องหมายทางคณิตศาสตร์ และกลุ่มของเครื่องหมายการเปรียบเทียบตามลำดับ ซึ่งก็เช่นเดียวกันคือ เครื่องหมายในกลุ่มนี้ถ้าหากมีใช้ในนิพจน์ การคำนวณก็จะกระทำจากทางซ้ายไปขวา แต่ในคลิปเปอร์มีการคำนวณหาค่าทางลอจิกที่พิเศษอีกอย่างหนึ่งที่เราเรียกว่า "Short-circuit evaluation" ซึ่งการคำนวณอาจจะหยุดก่อนที่จะจบนิพจน์ก็ได้ ถ้าหากเห็นแล้วว่าผลลัพธ์ของนิพจน์นั้นจะได้อะไร ดังรูปที่ 8

```
#define LIMIT 10
PRIVATE arr [LIMIT]
   .
   .
   .
i = 1
DO WHILE i <= LIMIT .AND. arr[i] < max
   process ( )
   i++
ENDDO
```

รูปที่ 8

เมื่อทำการคำนวณจนถึงรอบที่ i มีค่าเป็น 11 จะเห็นว่าในนิพจน์เงื่อนไขหลังคำสั่ง DO WHILE ในส่วนหน้าคือ i <= LIMIT จะทำให้คลิปเปอร์หยุดการคำนวณของนิพจน์เพียงเท่านี้ ไม่ต้องทำการตรวจสอบเงื่อนไขในส่วนหลังอีก ซึ่งจะช่วยให้การทำงานของโปรแกรมคลิปเปอร์ทำงานได้เร็วขึ้น

ในการคำนวณจากกลุ่มเครื่องหมายดังกล่าวข้างต้นทั้งหมด จะทำการคำนวณจากทางซ้ายไปขวามือของนิพจน์เสมอ ความจริงแล้วลำดับการคำนวณไม่มีผลสำคัญเท่าใดนัก แต่สำหรับในกรณีต่อไปนี้ เรื่องลำดับความสำคัญของการคำนวณก็มีผลดังตัวอย่าง

```
IF a( ) < b( )
```

โดยที่ a และ b เป็นฟังก์ชัน ซึ่งทั้ง a และ b อาจจะส่งผลกระทบข้างเคียงกันได้โดยที่เราคิดไม่ถึง เรียกว่า "Side effects" เช่นในกรณีนี้ถ้าทั้ง a และ b ใช้ตัวแปรร่วมกันและทั้งสองก็มีการเปลี่ยนแปลงค่าตัวแปรนั้นด้วย ในลักษณะนี้จะเห็นว่าเรื่องลำดับการคำนวณเริ่มมีความสำคัญ ซึ่งก็เหมือนกับตัวอย่างข้างล่างนี้

```
IF a[++i] < b[i++]
   .
   .
```

เครื่องหมายการกระทำในคลิปเปอร์มีลักษณะเป็นโอเวอร์โหลด (overload) กล่าวคือ สามารถใช้งานเครื่องหมายชนิดเดียวกันกับข้อมูลหลาย ๆ ชนิดได้ และบางครั้งเครื่องหมายชนิดเดียวกันก็มีหน้าที่ที่แตกต่างกัน เช่น เราสามารถใช้เครื่องหมายบวก (+) ได้หลายหน้าที่ ดังนี้คือ ใช้ในการบวกเลข ใช้เชื่อมข้อความ หรือใช้บวกวันที่ (date) เป็นต้น และเครื่องหมายลบ (-) บางครั้งก็ทำหน้าที่ในการลบเลข บางครั้งก็หมายถึงค่าติดลบของตัวเลขนั้น เป็นต้น

ดังนั้นในคลิปเปอร์จึงได้เพิ่มความหมายในการกำหนดค่าพิเศษขึ้นมาอีกเครื่องหมายหนึ่งคือ เครื่องหมาย ":=" ซึ่งจะหมายถึงเป็นการกำหนดค่าเท่านั้น ส่วนเครื่องหมาย "=" เป็นโอเวอร์โหลดคือ สามารถทำหน้าที่ทั้งเป็นการกำหนดค่าก็ได้ หรือถ้าถูกใช้ในนิพจน์ก็จะทำหน้าที่ในการเปรียบ

การคำนวณด้วยก็ให้ใช้เครื่องหมาย := แทนเครื่องหมาย = ดังตัวอย่าง

```
.IF (__rec__count := rec-
count ( )) > = 10000
```

ตัวแปร __rec__count จะถูกกำหนดค่าให้ก่อนจากค่าของฟังก์ชัน reccount ( ) แล้วจึงนำค่าดังกล่าวไปเปรียบเทียบกับ 10000 อีกที ซึ่งเสมือนกับว่าใช้คำสั่งดังนี้

```
__rec__count = reccount ( )
IF __rec__count > = 10000
.
.
.
```

ถ้าสังเกตจากตัวอย่างจะเห็นว่าในการใช้เครื่องหมาย := (ในคลิปเปอร์เรียกเครื่องหมายนี้ว่า inline assignment) นั้น เราจะใส่วงเล็บเอาไว้เพื่อทำการกำหนดค่าให้กับตัวแปร __rec__count ก่อน ทั้งนี้เพราะว่าลำดับการคำนวณของเครื่องหมาย := จะต่ำกว่ากลุ่มเครื่องหมายทางคณิตศาสตร์ ต่ำกว่ากลุ่มเครื่องหมายการเปรียบเทียบ และต่ำกว่ากลุ่มเครื่องหมายทางลอจิกตามลำดับ

เทคนิคบางประการ

ในการใช้เครื่องหมาย := เพื่อกำหนดค่าในนิพจน์ให้ทำการใส่วงเล็บลงไปด้วย เพื่อแก้ไขปัญหาเรื่องลำดับความสำคัญของการคำนวณ

เรื่องที่สำคัญอีกอย่างหนึ่งในการใช้เครื่องหมาย := ก็คือ ถ้ามีการใช้เครื่องหมาย := ในนิพจน์หลายที่การทำงานจะทำจากขวามาซ้าย (ต่างไปจากเครื่องหมายอื่น) เช่น

```
first__name := Last__name
:= zip := tel := space(10)
```

จะมีการทำงานดังนี้

```
first__name := (Last__name
:= (zip := (tel := space(10))))
```

ข้อดีของการใช้เครื่องหมาย := ดังที่ได้กล่าวแล้วก็คือ สามารถใช้ในนิพจน์ได้ ซึ่งเป็นเรื่องที่มีประโยชน์มาก ดังรูปที่ 9

จากคำสั่งในรูปที่ 9 จะพบว่าหลังจากออกจากลูปการทำงานของคำสั่ง DO WHILE แล้ว ถ้าเราต้องการทราบว่าค่าของ Valid__Key( ) คืออะไรที่ทำให้ออกจากลูปการทำงาน เราก็จะต้องทำการเรียกใช้ฟังก์ชัน Valid__Key ( ) อีก ซึ่งไม่จำเป็นเลยถ้าเราใช้เครื่องหมาย := ช่วยในการเขียนโปรแกรมจะช่วยให้เขียนคำสั่งใหม่ได้ ดังรูปที่ 10

ซึ่งจะเห็นว่าเราไม่จำเป็นต้องเรียกใช้ฟังก์ชัน valid__key( ) มาคำนวณให้เสียเวลาอีก เมื่อต้องการตรวจสอบว่าโปรแกรมหลุดออกจากลูปการทำงานได้อย่างไร เราสามารถจะทำการตรวจสอบค่าได้จากตัวแปร vk หรือตัวแปร ir ได้เลยทันที

เครื่องหมายอีกชนิดหนึ่งที่จะกล่าวถึงก็คือ เครื่องหมาย $ ซึ่งใช้สำหรับตรวจสอบว่าข้อความอันหนึ่งมีปนอยู่ในข้อความอื่นหรือไม่ เช่น ต้องการตรวจสอบว่าตัวอักษรที่เก็บอยู่ในตัวแปร part__type เป็นตัวอักษร R,D หรือ B สามารถนำเอา $ มาช่วยในการตรวจสอบได้ดังนี้

```
IF part__type $ "RDB"
.
.
```

หรือถ้าต้องการตรวจสอบตัว

```
DO WHILE Valid__Key( ) .AND. in__range( )
   .
   .
   .
ENDDO
IF Valid__Key( )
   .
   .
   .
ENDIF
```

รูปที่ 9

```
DO WHILE (vk := valid__key( )) .AND. (ir:=
     in__range( ))
   .
   .
   .
ENDDO
IF vk
   .
   .
ENDIF
```

รูปที่ 10

อักษรสองตัวที่เก็บอยู่ในตัวแปรว่าเป็น "CA", "NV" หรือ "AZ" ก็สามารถเขียนได้ดังนี้

```
IF state $ "AZ_CA_NV"
```

นิพจน์เงื่อนไข

นิพจน์เงื่อนไข IF/ELSE/ENDIF นั้นสามารถเขียนให้อยู่ในรูปแบบที่สั้นกะทัดรัดได้ โดยใช้นิพจน์เงื่อนไข IIF แทนได้ ดังตัวอย่าง ถ้าต้องการหาค่าที่น้อยที่สุดระหว่างเลขสองจำนวน คือ a และ b สามารถใช้นิพจน์เงื่อนไข IIF ได้ดังนี้

```
smaller = iif (a < b, a, b)
```

ซึ่งจะมีค่าเท่ากับเขียนด้วยคำสั่ง IF/ELSE/ENDIF ดังนี้

```
IF a<b
   smaller = a
ELSE
   smaller = b
ENDIF
```

ข้อดีของการใช้นิพจน์เงื่อนไข IIF ก็คือ มันสามารถจะนำไปใช้ในนิพจน์ใด ๆ ก็ได้ตามที่เราต้องการ โดยเฉพาะในเรื่องของมาโครคอมพายล์และโค้ดบล็อก (code block) หรือใช้ในคำสั่งที่ให้มีการทำงานแบบแบตช์ (batch) ซึ่งคลิปเปอร์ยอมให้มีการกำหนดนิพจน์เงื่อนไขของ IIF ที่ซับซ้อนได้ โดยให้มีการใช้เครื่องหมายวงเล็บ ( ) ช่วย ซึ่งสิ่งสำคัญจากการใช้นิพจน์เงื่อนไข IIF ก็คือ จะต้องมีการส่งผลลัพธ์กลับมา ดังตัวอย่างถ้าต้องการจะขอดูรายละเอียดของข้อมูลในแฟ้มข้อมูลจากฟิลด์สามฟิลด์ โดยจะให้มีการหยุดแสดงผลเมื่อครบทุก ๆ 24 บรรทัด เพื่อรอคอยการกดคีย์เสียก่อน ดังรูปที่ 11

```
rec_num = 0
LIST f_name, l_name, add1, iif (+ + rec_num;
                                % 24 != 0, " ",;
                                (qout("Press any key");
                                inkey(0), " "))
```

รูปที่ 11

```
rec_num = 0
REPORT FORM test WHILE iff(+ +rec_num % 24
! = O, .T., (qout ("Press any key"), inkey (O), .T.));
```

รูปที่ 12

จากนิพจน์ LIST จะเห็นว่าผลลัพธ์ในส่วนที่ 4 คือ เงื่อนไข iif นั้นจะแสดงค่าเป็นอะไรย่อมขึ้นอยู่กับค่าของตัวแปร rec_num โดยเงื่อนไข iif จะทำการตรวจสอบเงื่อนไขโดยใช้เครื่องหมาย % ซึ่งเป็นการหาค่า modulo ระหว่างค่าของตัวแปร rec_num กับ 24 (โดยนำค่าของตัวแปร rec_num ตั้งหารด้วย 24 ผลลัพธ์ที่ได้จากค่า Modulo คือเศษที่เหลือ) ถ้าผลลัพธ์ที่ได้ยังไม่เท่ากับ 0 แสดงว่าเงื่อนไขเป็นจริง ดังนั้นผลลัพธ์ที่ได้จากเงื่อนไข iif จะได้เป็นค่าว่างเปล่า (null string) กลับมา แต่ถ้าค่าของ rec_num เป็น 24 ดังนั้นผลลัพธ์จากการหาค่า modulo จะได้ 0 ซึ่งจะทำให้เงื่อนไขเป็นเท็จ ดังนั้นผลลัพธ์ที่ได้จากเงื่อนไข iif ก็คือจะแสดงข้อความว่า "Press any key" ขึ้นในบรรทัดที่ 24 แล้วรอคอยให้เราทำการกดคีย์ใด ๆ พร้อมกันนั้นก็จะส่งค่าว่างเปล่า (null string) กลับมาเหมือนเดิม (สังเกตการใช้วงเล็บช่วยในการกำหนดเงื่อนไขที่ซับซ้อน)

เราสามารถจะใช้เทคนิคอันเดียวกันนี้กับคำสั่งแบบแบตช์ (batch) อื่น ๆ ได้เช่นเดียวกัน แต่สิ่งสำคัญที่เราจะต้องให้ความสนใจเป็นพิเศษก็คือ ค่าหรือผลลัพธ์ที่จะส่งกลับมาจากการใช้เงื่อนไข iif นั้น ๆ

ดังตัวอย่างถ้าต้องการให้แสดงผลตามรูปแบบรายงานที่เราได้สร้างเอาไว้ด้วยคำสั่ง CREATE REPORT โดยจะทำให้ผลลัพธ์ที่แสดงนั้นหยุดแสดงชั่วคราวเมื่อครบทุก ๆ 24 บรรทัด เพื่อรอคอยการกดคีย์จากเราก่อนจึงจะแสดงผลต่อไปได้ ซึ่งมีการใช้คำสั่งในการวนลูปคือ WHILE ดังนั้นผลลัพธ์ที่จะถูกส่งกลับมาจากการใช้เงื่อนไข iif นั้นจะต้องได้ค่าจริง (.T.) เสมอ เพื่อทำให้คำสั่ง WHILE สามารถจะดำเนินการต่อไปได้โดยไม่หยุดชะงัก โดยมีการใช้คำสั่งดังรูปที่ 12

รูปแบบการกำหนดการเลือกทำ (Branching)

ในคลิปเปอร์มีรูปแบบคำสั่งที่ใช้ในการกำหนดคำสั่งย่อย ๆ ได้ 2 รูปแบบ คือ

1. IF/ELSE/ELSEIF/ENDIF
2. DO CASE/ENDCASE

ในคลิปเปอร์ไม่มีคำสั่งที่ใช้ในการกระโดดข้าม (JUMP) เลย จะมีแต่รูปแบบในคำสั่งการเลือกทำเท่านั้น

ELSEIF เป็นคำสั่งที่สั้นกะทัดรัดขึ้น ซึ่งมีค่าเท่ากับการเขียนด้วยคำสั่งบล็อกของ IF/ENDIF นั่นเอง ดังตัวอย่าง

```
IF X < 10
   .
   .
ELSE
   IF X > 20
      .
      .
   ENDIF
ENDIF
```

สามารถเขียนด้วยคำสั่ง ELSEIF ดังนี้

```
IF X < 10
   .
   .
ELSEIF X >20
   .
   .
ENDIF
```

แต่ถ้าต้องการทดสอบเงื่อนไขหลาย ๆ เงื่อนไข ประโยคคำสั่งที่เหมาะสมที่จะใช้ก็คือ CASE ซึ่งจะเหมือนกับการใช้คำสั่ง IF/ELSEIF หลาย ๆ ครั้งนั่นเอง การทำงานของคำสั่ง CASE ในคลิปเปอร์นั้นมันจะทำการตรวจสอบเงื่อนไขของแต่ละ CASE จนกว่าจะพบเงื่อนไขของ CASE ที่ให้ผลลัพธ์เป็นค่าจริงจึงจะทำงานตามคำสั่งที่อยู่ต่อจากคำสั่ง CASE อันนั้น จนกว่าจะถึงคำสั่ง CASE อันต่อไป หรือถึงคำสั่ง ENDCASE หรือถึงคำสั่ง OTHERWISE จึงจะหยุดการทำงาน แล้วจะกระโดดข้ามไปยังประโยคคำสั่ง ENDCASE เพื่อจบการทำงานของการใช้คำสั่ง DO CASE นั้น ๆ

OTHERWISE เป็นวลีที่จะใช้เป็นคำสั่งท้ายสุดต่อจาก CASE อันสุดท้ายของบล็อก DO CASE/ENDCASE ใช้สำหรับกรณีเมื่อทำการตรวจสอบเงื่อนไขถาม CASE แต่ละ CASE แล้วปรากฏว่าไม่ตรงกับ CASE ใด ๆ เลยก็จะให้แสดงผลลัพธ์ตามประโยคที่อยู่หลังคำสั่ง OTHERWISE จนกว่าจะพบประโยคคำสั่ง ENDCASE จึงจะหยุดการทำงาน

รูปแบบ BEGIN SEQUENCE/END

รูปแบบ BEGIN SEQUENCE/END นั้นใช้เพื่อควบคุมบล็อกการทำงานของ IF/ENDIF ที่ใด ๆ ก็ตาม เพื่อให้จบการทำงานรวดเร็วขึ้น (คือให้มาทำงานตามคำสั่งที่อยู่หลังประโยค END ทันที) โดยการใช้คำสั่ง BREAK แทรกลงไปในบล็อกที่เราต้องการ ดังรูปที่ 13

หรือก่อนจะถึงประโยคคำสั่ง END ถ้าเรามีการใช้คำสั่ง RECOVER อยู่ก่อน ดังนั้นในการใช้คำสั่ง BREAK จะส่งผลให้โปรแกรมกระโดดข้ามขั้นตอนการทำงานไปทำตามคำสั่งที่อยู่ถัดจากคำสั่ง RECOVER ทันที แล้วจึงกระโดดไปทำงานในคำสั่งที่อยู่ถัดจากประโยค END อีกทีหนึ่ง ดังรูปที่ 14

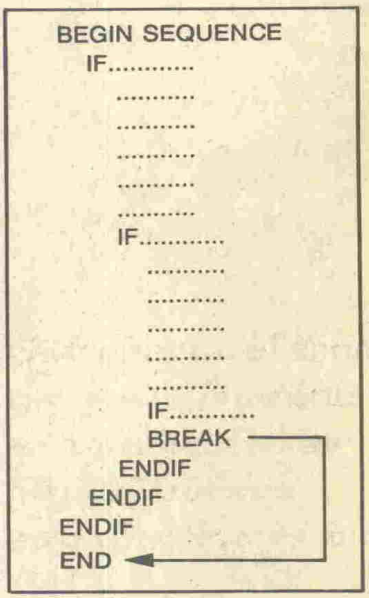
เราสามารถจะทำการส่งค่าจากคำสั่ง BREAK ไปยังคำสั่ง RECOVER ได้ โดยเราเขียนคำสั่ง BREAK ดังนี้

```
BREAK error__val
```

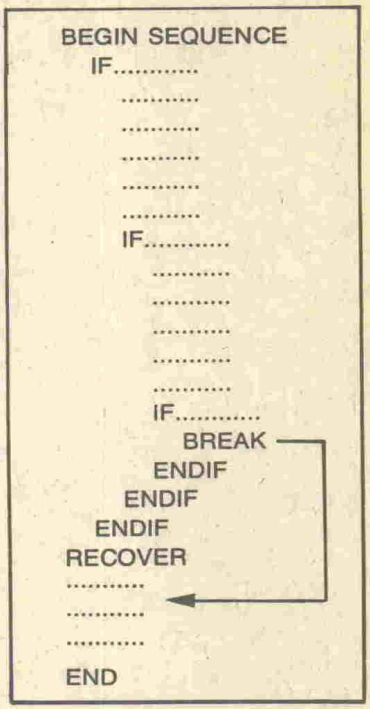
และเขียนคำสั่ง RECOVER ดังนี้

```
RECOVER USING rec__val
```

ดังนั้นตัวแปร rec__val จะถูกกำหนดค่าให้โดยตัวแปร error__val ซึ่งถ้าเรามีการใช้คำสั่ง BREAK เพื่อควบคุมโปรแกรมในหลาย ๆ จุด และได้ทำการกำหนดค่าคำสั่ง BREAK ในแต่ละจุดให้แตกต่างกัน คำสั่ง RECOVER ก็สามารถทราบได้ว่าสาเหตุที่ทำให้โปรแกรมหยุดการทำงานด้วยคำสั่ง BREAK นั้น เนื่องมาจากการใช้คำสั่ง BREAK ที่จุดใด โดยดูจากค่าของตัวแปรที่ตามหลังคำสั่ง RECOVER



รูปที่ 13



รูปที่ 14